PPL- UNIT 4

Object-Oriented Programming

A language that is object oriented must provide support for three key language features:

- Abstract data types
- Inheritance
  - Inheritance is the central theme in OOP and languages that support it
- dynamic binding of method calls to methods

Inheritance

- The process of acquiring properties from base class to subclass
- The syntactic form of a derived class is

  class derived_class_name : derivation_mode base_class_name
  {data member and member function declarations};

  The derivation_mode can be either **public** or **private**.

As a simple example of inheritance, consider the following: Suppose we have a class named Vehicles, which has variables for year, color, and make. A natural specialization, or subclass, of this would be Truck, which could inherit the variables from Vehicle, but would add variables for hauling capacity and number of wheels. Figure 12.1 shows a simple diagram to indicate the relationship between the Vehicle class and the Truck class, in which the arrow points to the parent class.

There are several ways a derived class can differ from its parent. Following are the most common differences between a parent class and its subclasses:

1. The parent class can define some of its variables or methods to have private access, which means they will not be visible in the subclass.
2. The subclass can add variables and/or methods to those inherited from the parent class.
3. The subclass can modify the behavior of one or more of its inherited methods. A modified method has the same name, and often the same protocol, as the one of which it is a modification. The new method is said to **override** the inherited method, which is then called an **overridden method**.

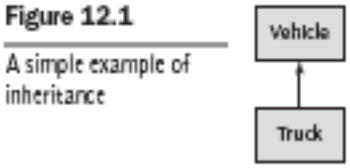


Figure 12.1
A simple example of inheritance

Classes can have two kinds of methods and two kinds of variables. The most commonly used methods and variables are called **instance methods** and **instance variables**. Every object of a class has its own set of instance variables, which store the object's state. The only difference

between two objects of the same class is the state of their instance variables. For example, a class for cars might have instance variables for color, make, model, and year. Instance methods operate only on the objects of the class. **Class variables** belong to the class, rather than its object, so there is only one copy for the class. For example, if we wanted to count the number of instances of a class, the counter could not be an instance variable—it would need to be a class variable

- Productivity increases can come from reuse
  - ADTs are difficult to reuse—always need changes
  - All ADTs are independent and at the same level
- Inheritance allows new classes defined in terms of existing ones, i.e., by allowing them to inherit common parts
- ADTs are usually called *classes*.Class instances are called *objects*
- A class that inherits is a *derived class* or a *subclass*
- The class from which another class inherits is a parent class or *superclass*
- In the simplest case, a class inherits all of the entities of its parent
- Inheritance can be complicated by access controls to encapsulated entities
- A class can hide entities from its subclasses
- A class can hide entities from its clients
- A class can also hide entities for its clients while allowing its subclasses to see them
- Besides inheriting methods as is, a class can modify an inherited method
- There are two kinds of methods in a class:
  - *Class methods* – accept messages to the class
  - *Instance methods* – accept messages to objects
- One disadvantage of inheritance for reuse:
  Creates interdependencies among classes that complicate maintenance

  Inheritance Example in C++

```
class base_class
{
  private:
    int a;
    float x;
  protected:
    int b;
    float y;
  public:
    int c;
    float z;
};
class subclass_1 : public base_class { … };
//     In this one, b and y are protected and
//     c and z are public
class subclass_2 : private base_class { … };
//    In this one, b, y, c, and z are private,
```

```
// and no derived class has access to any
// member of base_class
```

Abstract data types

Data hiding is known as encapsulation. It is a procedure of forming objects. An encapsulated object is often called as an Abstract data type. We need to encapsulate data because programmer often makes mistakes and data get changed accidentally. Thus to protect data we need to construct a secure wall to protect the data. Data hiding is nothing but making data variable of the class **private.** C++ classes are abstract data types. A C++ program unit that declares an instance of a class can also access any of the public entities in that class, but only through an instance of the class. This is a cleaner and more direct way to provide abstract data types

Dynamic binding

Member functions that must be dynamically bound must be declared to be virtual functions by preceding their headers with the reserved word **virtual**, which can appear only in a class body. A method can be defined to be virtual, which means that they can be called through polymorphic variables and dynamically bound to messages. A pointer variable that has the type of a base class can be used to point to any heap-dynamic objects of any class publicly derived from that base class, making it a polymorphic variable. A *polymorphic variable* can be defined in a class that is able to reference (or point to) objects of the class and objects of any of its descendants C++ does not allow value variables to be polymorphic. A pure virtual function has no definition at all . A class that has at least one pure virtual function is an *abstract class*

Example of dynamic binding

```
class Shape
{
public:
virtual void draw() = 0;
. . .
};
class Circle : public Shape
 {
public:
void draw() { . . . }
. . .
};
class Rectangle : public Shape
{
public:
void draw() { . . . }
. . .
};
class Square : public Rectangle
{
public:
void draw() { . . . }
. . .
};
```

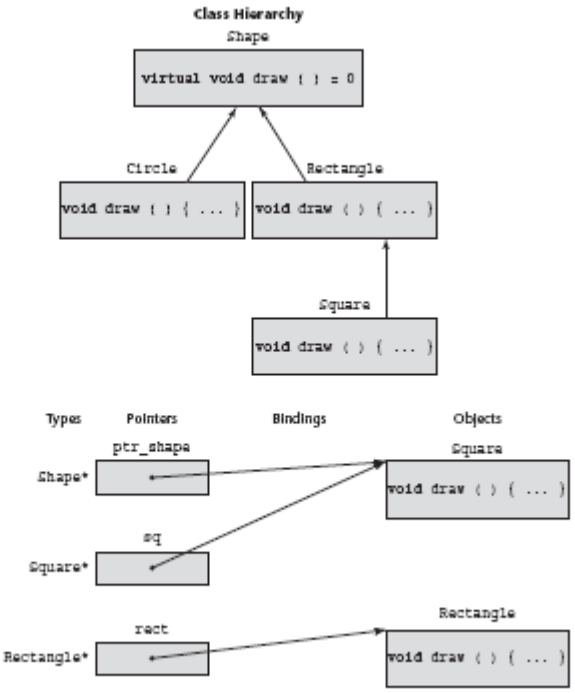
Given these definitions, the following code has examples of both statically and dynamically bound calls:

```
Square* sq = new Square;
```

```
Rectangle* rect = new Rectangle;
Shape* ptr_shape;
ptr_shape = sq; // Now ptr_shape points to a
// Square object
ptr_shape->draw(); // Dynamically bound to the draw
// in the Square class
rect->draw(); // Statically bound to the draw
// in the Rectangle class
```

This situation is shown in Figure 12.6.

Figure 12.6
Dynamic binding



dynamic binding allows the code that uses members like `draw` to be written before all or even any of the versions of `draw` are written. New derived classes could be added years later, without requiring any change to the code that uses such dynamically bound members. This is a highly useful feature of object-oriented languages.

Uses of dynamic binding

Abstract classes and inheritance together support a powerful technique for software development.

Dynamic binding Allows software systems to be more easily extended during both development and maintenance e.g., new shape classes are defined later

Design Issues for OOP Languages

- The exclusivity of objects
- Are subclasses subtypes?
- Type checking and polymorphism

- Single and multiple inheritance
- Object allocation and deallocation
- Dynamic and static binding
- Nested classes
- Initialization of objects

The exclusivity of objects

- Everything is an object
  - Advantage - elegance and purity
  - Disadvantage - slow operations on simple objects
- Add objects to a complete typing system
  - Advantage - fast operations on simple objects
  - Disadvantage - results in a confusing type system (two kinds of entities)
- Include an imperative-style typing system for primitives but make everything else objects
  - Advantage - fast operations on simple objects and a relatively small typing system
  - Disadvantage - still some confusion because of the two type systems

Are subclasses subtypes?

- Does an "is-a" relationship hold between a parent class object and an object of subclass?
  - If a derived class is-a parent class, then objects of derived class behave same as parent class object
    subtype small_Int is Integer range 0 .. 100;
  - Every **small_Int** variable can be used anywhere **Integer** variables can be used
- A derived class is a subtype if methods of subclass that override parent class are type compatible with the overridden parent methods
  - Subclass can only add variables and methods and override inherited methods in "compatible" ways

Type Checking and Polymorphism

- Polymorphism may require dynamic type checking of parameters and the return value
  - Dynamic type checking is costly and delays error detection
- If overriding methods are restricted to having the same parameter types and return type, the checking can be static
- Dynamic type checking is costly and delays error detection

Single and Multiple inheritance

- Multiple inheritance allows a new class to inherit from two or more classes
- Disadvantages of multiple inheritance:
  - Language and implementation complexity (in part due to name collisions)
  - Potential inefficiency - dynamic binding costs more with multiple inheritance (but not much)
- Advantage:

- Sometimes it is quite convenient and valuable

Object Allocation and Deallocation

- From where are objects allocated?
  - If behave like ADTs, can be allocated from anywhere
    - Allocated from the run-time stack
    - Explicitly created on the heap (via **new**)
  - If they are all heap-dynamic, references can be uniform thru a pointer or reference variable simplifies assignment
  - If objects are stack dynamic, assignment of subclass B's object to superclass A's object is value copy, but what if B is larger in space?
- Is deallocation explicit or implicit?

Dynamic and Static Binding

- Should all binding of messages to methods be dynamic?
  - If none are, you lose the advantages of dynamic binding
  - If all are, it is inefficient
- Alternative: allow the user to specify

Nested Classes

- If a new class is needed by only one class, there is no reason to define so it can be seen by other classes
  - Can the new class be nested inside the class that uses it?
  - In some cases, the new class is nested inside a subprogram rather than directly in another class
- Other issues:
  - Which facilities of the nesting class should be visible to the nested class and vice versa

12.11 Implementation of Object-Oriented Constructs

There are at least two parts of language support for object-oriented programming

- storage structures for instance variables (Instance Data Storage)
- dynamic bindings of messages to methods

12.11.1 Instance Data Storage

In C++, classes are defined as extensions of C's record structures—structs. This similarity suggests a storage structure for the instance variables of class instances—that of a record. This form of this structure is called a **class instance record (CIR)**. The structure of a CIR is static, so it is built at compile time and used as a template for the creation of the data of class instances. Every class has its own CIR. When a derivation takes place, the CIR for the subclass is a copy of that of the parent class, with entries for the new instance variables added at the end. Because the structure of the CIR is static, access to all instance variables can be done as it is in records, using constant offsets from the beginning of the CIR instance. This makes these accesses as efficient as those for the fields of records.

12.11.2 Dynamic Binding of Method Calls to Methods

- Methods in a class that are statically bound need not be involved in the CIR; methods that will be dynamically bound must have entries in the CIR
  - Calls to dynamically bound methods can be connected to the corresponding code thru a pointer in the CIR
  - The storage structure is sometimes called *virtual method tables* (vtable)
  - Method calls can be represented as offsets from the beginning of the vtable

Consider the following Java example, in which all methods are dynamically bound:

```
public class A
{
public int a, b;
public void draw() { . . . }
public int area() { . . . }
}
public class B extends A
{
public int c, d;
public void draw() { . . . }
public void sift() { . . . }
}
```

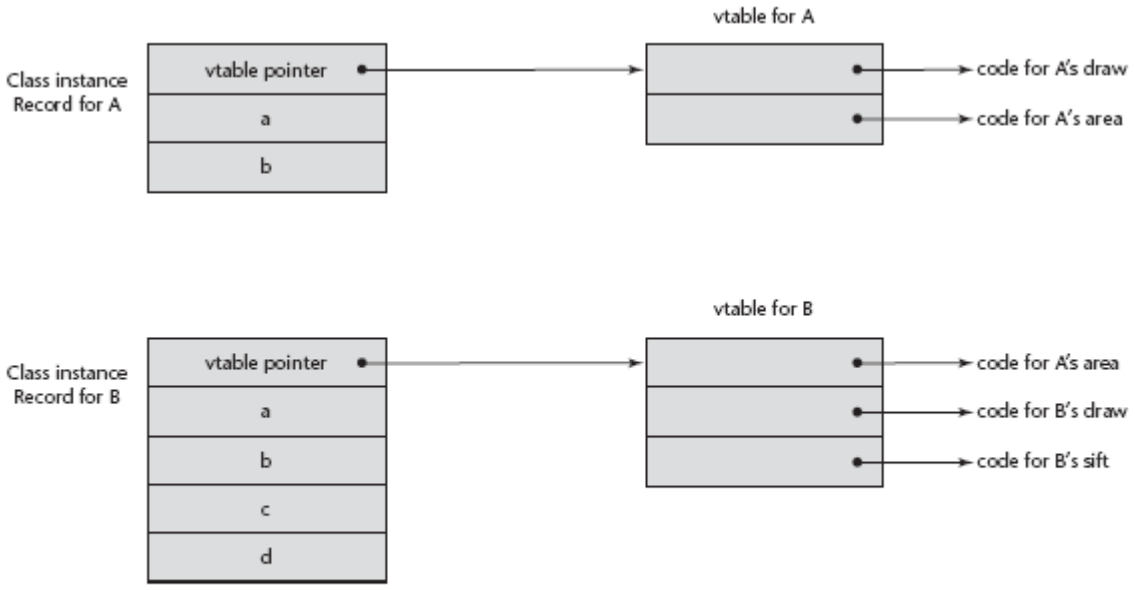


Figure 12.7

An example of the CIRs with single inheritance

The CIRs for the A and B classes, along with their vtables, are shown in Figure 12.7. Notice that the method pointer for the area method in B's vtable points to the code for A's area method. The reason is that B does not override A's area method, so if a client of B calls area, it is the area method inherited from A. On the other hand, the pointers for draw and sift in B's vtable point to B's draw and sift. The draw method is overridden in B and sift is defined as an addition in B.

Multiple inheritance complicates the implementation of dynamic binding.

Consider the following three C++ class definitions:

```
class A {
public:
int a;
virtual void fun() { . . . }
virtual void init() { . . . }
};
class B {
public:
int b;
virtual void sum() { . . . }
};
class C : public A, public B {
public:
int c;
virtual void fun() { . . . }
virtual void dud() { . . . }
};
```

The C class inherits the variable a and the init method from the A class. It redefines the fun method, although both its fun and that of the parent class A are potentially visible through a polymorphic variable (of type A). From B, C inherits the variable b and the sum method. C defines its own variable, c, and defines an uninherited method, dud. A CIR for C must include A's data, B's data, and C's data, as well as some means of accessing all visible methods. Under single inheritance, the CIR would include a pointer to a vtable that has the addresses of the code of all visible methods.
There must also be two vtables: one for the A and C view and one for the B view. The first part of the CIR for C in this case can be the C and A view, which begins with a vtable pointer for the methods of C and those inherited from A, and includes the data inherited from A. Following this in C's CIR is the B view part, which begins with a vtable pointer for the virtual methods of B, which is followed by the data inherited from B and the data defined in C. The CIR for C is shown in Figure 12.8.

An example of a subclass CIR with multiple parents

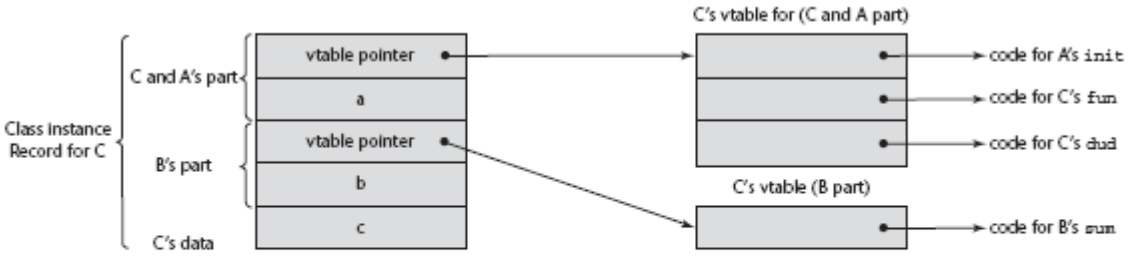


Figure 12.8

An example of a subclass CIR with multiple parents

13 Concurrency

Introduction

- Concurrency can occur at four levels:
  - Machine instruction level
  - High-level language statement level
  - Unit level
  - Program level
- Because there are no language issues in instruction- and program-level concurrency, they are not addressed here

Multiprocessor Architectures

- Late 1950s - one general-purpose processor and one or more special-purpose processors for input and output operations
- Early 1960s - multiple complete processors, used for program-level concurrency
- Mid-1960s - multiple partial processors, used for instruction-level concurrency
- Single-Instruction Multiple-Data (SIMD) machines
- Multiple-Instruction Multiple-Data (MIMD) machines
  - Independent processors that can be synchronized (unit-level concurrency)

Categories of Concurrency

- A *thread of control* in a program is the sequence of program points reached as control flows through the program
- Categories of Concurrency:
  - *Physical concurrency* - Multiple independent processors (multiple threads of control)
  - *Logical concurrency* - The appearance of physical concurrency is presented by time-sharing one processor (software can be designed as if there were multiple threads of control)
- Coroutines (*quasi-concurrency) have a single thread of control*

Motivations for Studying Concurrency

- Involves a different way of designing software that can be very useful—many real-world situations involve concurrency
- Multiprocessor computers capable of physical concurrency are now widely used

Introduction to Subprogram-Level Concurrency

- A *task* or *process* is a program unit that can be in concurrent execution with other program units
- Tasks differ from ordinary subprograms in that:
  - A task may be implicitly started
  - When a program unit starts the execution of a task, it is not necessarily suspended
  - When a task's execution is completed, control may not return to the caller
- Tasks usually work together

Two General Categories of Tasks

- *Heavyweight tasks* execute in their own address space
- *Lightweight tasks* all run in the same address space – more efficient

- A task is *disjoint* if it does not communicate with or affect the execution of any other task in the program in any way

Task Synchronization

- A mechanism that controls the order in which tasks execute
- Two kinds o f synchronization
  - *Cooperation* synchronization
  - *Competition* synchronization
- Task communication is necessary for synchronization, provided by:
  - Shared nonlocal variables
  - Parameters
  - Message passing

Kinds of synchronization

***Cooperation* synchronization**

- Task B must wait for task A to complete some specific activity before task B can continue its execution, e.g., the producer-consumer problem

***Competition* synchronization**

- Two or more tasks must use some resource that cannot be simultaneously used, e.g., a shared counter
  - Competition is usually provided by mutually exclusive access

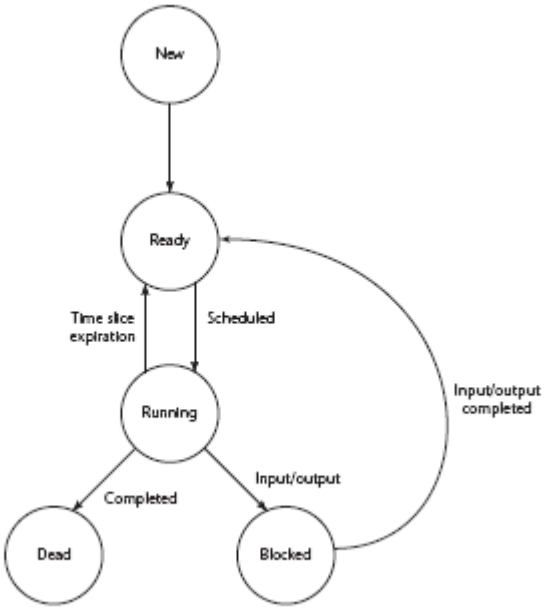
Scheduler

- Providing synchronization requires a mechanism for delaying task execution
- Task execution control is maintained by a program called the *scheduler*, which maps task execution onto available processors

Task Execution States

- *New* - created but not yet started
- *R*ready - ready to run but not currently running (no available processor)
- *Running*
- *Blocked* - has been running, but cannot now continue (usually waiting for some event to occur)
- *Dead* - no longer active in any sense

Figure 13.2
Flow diagram of task states



Liveness and Deadlock

- *Liveness* is a characteristic that a program unit may or may not have
  - In sequential code, it means the unit will eventually complete its execution
- In a concurrent environment, a task can easily lose its liveness
- If all tasks in a concurrent environment lose their liveness, it is called *deadlock*

Design Issues for Concurrency

- Competition and cooperation synchronization
- Controlling task scheduling
- How and when tasks start and end execution
- How and when are tasks created

Methods of Providing Synchronization

- Semaphores
- Monitors
- Message Passing

Semaphores

- Dijkstra - 1965
- A *semaphore* is a data structure consisting of a counter and a queue for storing task descriptors
- Semaphores can be used to implement guards on the code that accesses shared data structures
- Semaphores have only two operations, *wait* and *release* (originally called *P* and *V* by Dijkstra)
- Semaphores can be used to provide both competition and cooperation synchronization

Cooperation Synchronization with Semaphores

- Example: A shared buffer
- The buffer is implemented as an ADT with the operations DEPOSIT and FETCH as the only ways to access the buffer
- Use two semaphores for cooperation: emptyspots and fullspots
- The semaphore counters are used to store the numbers of empty spots and full spots in the buffer
- DEPOSIT must first check emptyspots to see if there is room in the buffer
- If there is room, the counter of emptyspots is decremented and the value is inserted
- If there is no room, the caller is stored in the queue of emptyspots
- When DEPOSIT is finished, it must increment the counter of fullspots
- FETCH must first check fullspots to see if there is a value
    - If there is a full spot, the counter of fullspots is decremented and the value is removed
    - If there are no values in the buffer, the caller must be placed in the queue of fullspots
    - When FETCH is finished, it increments the counter of emptyspots
- The operations of FETCH and DEPOSIT on the semaphores are accomplished through two semaphore operations named *wait* and *release*

Semaphores: Wait Operation

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
  decrement aSemaphore's counter
else
  put the caller in aSemaphore's queue
  attempt to transfer control to a ready task
    -- if the task ready queue is empty,
    -- deadlock occurs
end
```

Semaphores: Release Operation

```
release(aSemaphore)
if aSemaphore's queue is empty then
  increment aSemaphore's counter
else
  put the calling task in the task ready queue
  transfer control to a task from aSemaphore's queue
end
```

Cooperation Synchronization with semaphores

Producer Code

```
semaphore fullspots, emptyspots;
fullstops.count = 0;
```

```
emptyspots.count = BUFLEN;
task producer;
        loop
        -- produce VALUE –-
        wait (emptyspots); {wait for space}
        DEPOSIT(VALUE);
        release(fullspots); {increase filled}
        end loop;
end producer;
```

Consumer Code

```
task consumer;
        loop
        wait (fullspots);{wait till not empty}}
        FETCH(VALUE);
        release(emptyspots); {increase empty}
        -- consume VALUE –-
        end loop;
end consumer;
```

Competition Synchronization with Semaphores

- A third semaphore, named access, is used to control access (competition synchronization)
  - The counter of access will only have the values 0 and 1
  - Such a semaphore is called a *binary semaphore*
- Note that wait and release must be atomic!

producer code

```
semaphore access, fullspots, emptyspots;
access.count = 0;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
        loop
        -- produce VALUE –-
        wait(emptyspots); {wait for space}
        wait(access);     {wait for access)
        DEPOSIT(VALUE);
        release(access); {relinquish access}
        release(fullspots); {increase filled}
        end loop;
end producer;
```

Consumer Code

```
task consumer;
        loop
        wait(fullspots);{wait till not empty}
        wait(access);   {wait for access}
        FETCH(VALUE);
        release(access); {relinquish access}
        release(emptyspots); {increase empty}
        -- consume VALUE –-
        end loop;
end consumer;
```

Evaluation of Semaphores

- Misuse of semaphores can cause failures in cooperation synchronization, e.g., the buffer will overflow if the wait of fullspots is left out
- Misuse of semaphores can cause failures in competition synchronization, e.g., the program will deadlock if the release of access is left out

Monitors

One solution to some of the problems of semaphores in a concurrent environment is to encapsulate shared data structures with their operations and hide their representations—that is, to make shared data structures abstract data types with some special restrictions

- The idea: encapsulate the shared data and its operations to restrict access
- A monitor is an abstract data type for shared data

Competition Synchronization

- Shared data is resident in the monitor (rather than in the client units)
- All access resident in the monitor
  - Monitor implementation guarantee synchronized access by allowing only one access at a time
  - Calls to monitor procedures are implicitly queued if the monitor is busy at the time of the call

Cooperation Synchronization

- Cooperation between processes is still a programming task
  - Programmer must guarantee that a shared buffer does not experience underflow or overflow

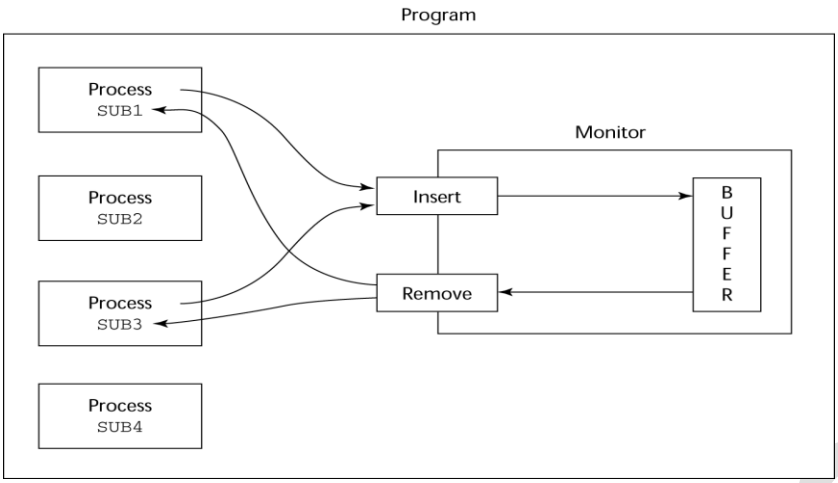


Fig: A program using a monitor to control access to a shared buffer

Evaluation of Monitors

- A better way to provide competition synchronization than are semaphores
- Semaphores can be used to implement monitors
- Monitors can be used to implement semaphores
- Support for cooperation synchronization is very similar as with semaphores, so it has the same problems

Message Passing

- Message passing is a general model for concurrency
  - It can model both semaphores and monitors
  - It is not just for competition synchronization
- Central idea: task communication is like seeing a doctor--most of the time she waits for you or you wait for her, but when you are both ready, you get together, or *rendezvous*

Message Passing Rendezvous

- To support concurrent tasks with message passing, a language needs:

- A mechanism to allow a task to indicate when it is willing to accept messages

-If task A is waiting for a message at the time task B sends that message, the message can be transmitted. This actual transmission of the message is called a **rendezvous**.

ADA SUPPORT FOR CONCURRENCY

- The Ada 83 Message-Passing Model
  - Ada tasks have specification and body parts, like packages; the spec has the interface, which is the collection of entry points:

```
task Task_Example is
        entry ENTRY_1 (Item : in Integer);
end Task_Example;
```

Task Body

- The body task describes the action that takes place when a rendezvous occurs
- A task that sends a message is suspended while waiting for the message to be accepted and during the rendezvous
- Entry points in the spec are described with accept clauses in the body

Example of a Task Body

```
task body Task_Example is
   begin
   loop
   accept Entry_1 (Item: in Float) do
   ...
   end Entry_1;
   end loop;
  end Task_Example;
```

The task executes to the top of the accept clause and waits for a message
During execution of the accept clause, the sender is suspended

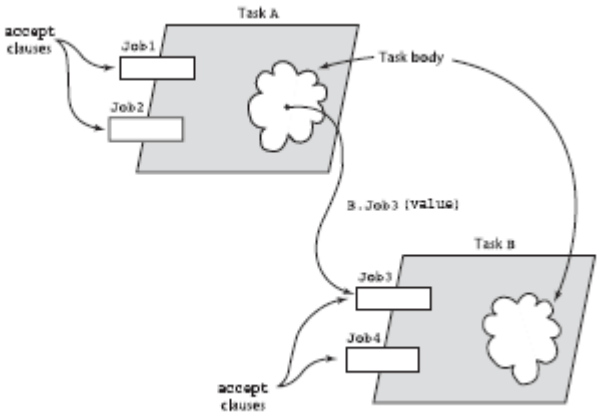


Figure 13.5
Graphical representation of a rendezvous caused by a message sent from task A to task B

The Concept of Synchronous Message Passing

Message passing can be either synchronous or asynchronous.

The basic concept of synchronous message passing is that tasks are often busy, and when busy, they cannot be interrupted by other units. Suppose task `A` and task `B` are both in execution, and `A` wishes to

send a message to B. Clearly, if B is busy, it is not desirable to allow another task to interrupt it. That would disturb B's current processing. . The alternative is to provide a Linguistic mechanism that allows a task to specify to other tasks when it is ready to receive messages.
.
If task A is waiting for a message at the time task B sends that message, the message can be transmitted. This actual transmission of the message is called a **rendezvous**. Note that a rendezvous can occur only if both the sender and receiver want it to happen. During a rendezvous, the information of the message can be transmitted in either or both directions.

Asynchronous message passing

- Provided through asynchronous select structures
- An asynchronous select has two triggering alternatives, and entry clause or a delay - The entry clause is triggered when sent a message; the delay clause is triggered when its time limit is reached

```
task WATER_MONITOR; -- specification
task body WATER_MONITOR is -- body
begin
loop
if WATER_LEVEL > MAX_LEVEL
then SOUND_ALARM;
end if;
delay 1.0; -- No further execution
-- for at least 1 second
end loop;
end WATER_MONITOR;
```

Both cooperation and competition synchronization of tasks can be conveniently handled with the message-passing model, as described in the following section.

Cooperation Synchronization with message Passing
- Provided by Guarded accept clauses
-Guarded commands are the basis of the construct designed for controlling message passing.

- Example:

```
task body buf_task is
    begin
    loop

when not FULL(BUFFER) =>
accept DEPOSIT (NEW_VALUE) do

--------
end DEPOSIT;
end loop;
end buf_task;
```

Def: A clause whose guard is true is called open.
Def: A clause whose guard is false is called closed.

Competition Synchronization with Message Passing:
- Example--a shared buffer
- Encapsulate the buffer and its operations in a task
- Competition synchronization is implicit in the semantics of accept clauses
- Only one accept clause in a task can be active at any given time

Ex: The tasks for the producer and consumer that could use buf_task have the following form

```
task Buf_Task is
entry Deposit(Item : in Integer);
entry Fetch(Item : out Integer);
end Buf_Task;

task body Buf_Task is
Bufsize : constant Integer := 100;
begin
loop
select
when Filled < Bufsize =>
accept Deposit(Item : in Integer) do
Buf(Next_In) := Item;
end Deposit;
or
when Filled > 0 =>
accept Fetch(Item : out Integer) do
Item := Buf(Next_Out);
end Fetch;
Filled := Filled - 1;
end select;
end loop;
end Buf_Task;
```

In this example, both **accept** clauses are extended. These extended clauses can be executed concurrently with the tasks that called the associated **accept** clauses.The tasks for a producer and a consumer that could use `Buf_Task` have the following form:

```
task Producer;
task Consumer;
task body Producer is
New_Value : Integer;
begin
loop
-- produce New_Value --
Buf_Task.Deposit(New_Value);
end loop;
end Producer;

task body Consumer is
Stored_Value : Integer;
begin
```

```
loop
Buf_Task.Fetch(Stored_Value);
-- consume Stored_Value --
end loop;
end Consumer;
```

Java Threads

- The concurrent units in Java are methods named run
  - A run method code can be in concurrent execution with other such methods
  - The process in which the run methods execute is called a *thread*

```
Class myThread extends Thread
        public void run () {…}
}
…
Thread myTh = new MyThread ();
myTh.start();
```

Controlling Thread Execution

- The Thread class has several methods to control the execution of threads
  - The **yield** is a request from the running thread to voluntarily surrender the processor
  - The **sleep** method can be used by the caller of the method to block the thread
  - The **join** method is used to force a method to delay its execution until the run method of another thread has completed its execution

Thread Priorities

- A thread's default priority is the same as the thread that create it
  - If main creates a thread, its default priority is NORM_PRIORITY
- Threads defined two other priority constants, MAX_PRIORITY and MIN_PRIORITY
- The priority of a thread can be changed with the methods setPriority

Competition Synchronization with Java Threads

- A method that includes the synchronized modifier disallows any other method from running on the object while it is in execution

```
…
public synchronized void deposit( int i) {…}
public synchronized int fetch() {…}
…
```

- The above two methods are synchronized which prevents them from interfering with each other

- If only a part of a method must be run without interference, it can be synchronized thru synchronized statement

synchronized (*expression*)
statement

Cooperation Synchronization with Java Threads

- Cooperation synchronization in Java is achieved via wait, notify, and notifyAll methods
  - All methods are defined in Object, which is the root class in Java, so all objects inherit them
- The wait method must be called in a loop
- The notify method is called to tell one waiting thread that the event it was waiting has happened
- The notifyAll method awakens all of the threads on the object's wait list

Java's Thread Evaluation

- Java's support for concurrency is relatively simple but effective
- Not as powerful as Ada's tasks

C# Threads

- Loosely based on Java but there are significant differences
- Basic thread operations
  - Any method can run in its own thread
  - A thread is created by creating a Thread object
  - Creating a thread does not start its concurrent execution; it must be requested through the Start method
  - A thread can be made to wait for another thread to finish with Join
  - A thread can be suspended with Sleep
  - A thread can be terminated with Abort

Synchronizing Threads

- Three ways to synchronize C# threads
  - The Interlocked class
    - Used when the only operations that need to be synchronized are incrementing or decrementing of an integer
  - The lock statement
    - Used to mark a critical section of code in a thread

  lock (expression) {… }
  - The Monitor class
    - Provides four methods that can be used to provide more sophisticated synchronization

C#'s Concurrency Evaluation

- An advance over Java threads, e.g., any method can run its own thread
- Thread termination is cleaner than in Java
- Synchronization is more sophisticated

Statement-Level Concurrency

- Objective: Provide a mechanism that the programmer can use to inform compiler of ways it can map the program onto multiprocessor architecture
- Minimize communication among processors and the memories of the other processors

High-Performance Fortran

- A collection of extensions that allow the programmer to provide information to the compiler to help it optimize code for multiprocessor computers
- Specify the number of processors, the distribution of data over the memories of those processors, and the alignment of data

Primary HPF Specifications

- Number of processors
  !HPF$ PROCESSORS procs (n)
- Distribution of data
  !HPF$ DISTRIBUTE (*kind*) ONTO procs :: *identifier_list*
  - *kind* can be BLOCK (distribute data to processors in blocks) or CYCLIC (distribute data to processors one element at a time)
- Relate the distribution of one array with that of another
  ALIGN *array1_element* WITH *array2_element*

Statement-Level Concurrency Example

```
REAL list_1(1000), list_2(1000)
   INTEGER list_3(500), list_4(501)
!HPF$ PROCESSORS proc (10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs ::list_1, list_2
!HPF$ ALIGN list_1(index) WITH list_2 (index)
!HPF$ ALIGN list_3(index) WITH list_4 (index+1)
   …
   list_1 (index) = list_2(index)
   list_3(index) = list_4(index+1)
```

- FORALL statement is used to specify a list of statements that may be executed concurrently

```
FORALL (index = 1:1000)
list_1(index) = list_2(index)
```

- Specifies that all 1,000 RHSs of the assignments can be evaluated before any assignment takes place

Summary

- Concurrent execution can be at the instruction, statement, or subprogram level
- Physical concurrency: when multiple processors are used to execute concurrent units
- Logical concurrency: concurrent united are executed on a single processor
- Two primary facilities to support subprogram concurrency: competition synchronization and cooperation synchronization
- Mechanisms: semaphores, monitors, rendezvous, threads
- High-Performance Fortran provides statements for specifying how data is to be distributed over the memory units connected to multiple processors

Introduction to Exception Handling

- In a language without exception handling
  - When an exception occurs, control goes to the operating system, where a message is displayed and the program is terminated
- In a language with exception handling
  - Programs are allowed to trap some exceptions, thereby providing the possibility of fixing the problem and continuing

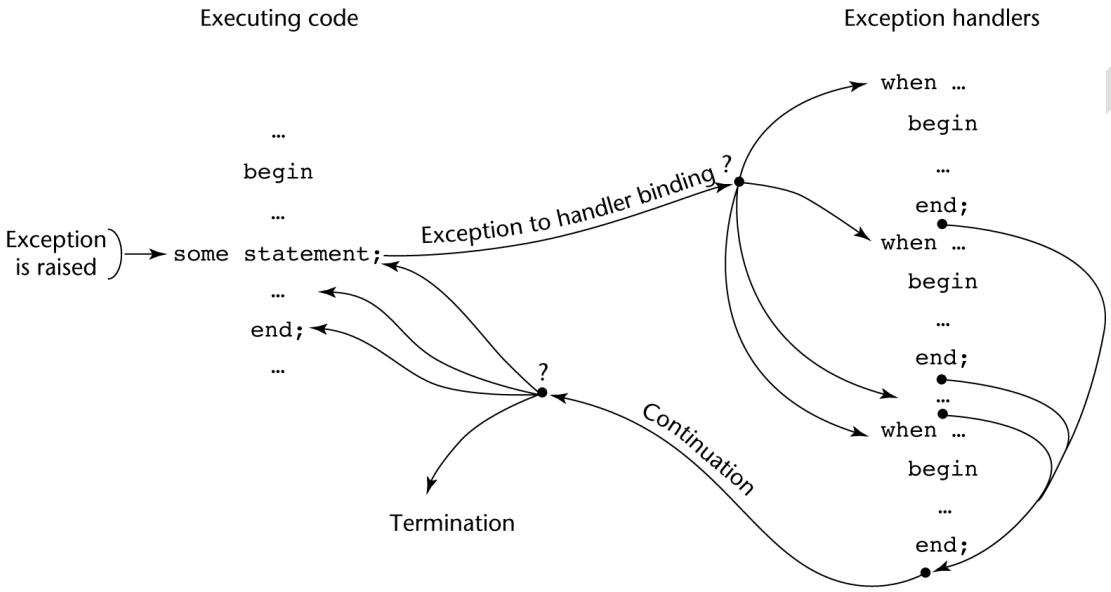
Basic Concepts

- Many languages allow programs to trap input/output errors (including EOF)
- An *exception* is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing
- The special processing that may be required after detection of an exception is called *exception handling*
- The exception handling code unit is called an *exception handler*

Advantages of Built-in Exception Handling

- Error detection code is tedious to write and it clutters the program
- Exception handling encourages programmers to consider many different possible error

Exception Handling Control Flow



Exception Handling in Ada

- The frame of an exception handler in Ada is either a subprogram body, a package body, a task, or a block
- Because exception handlers are usually local to the code in which the exception can be raised, they do not have parameters

Ada Exception Handlers

- Handler form:
  when *exception_choice*{|*exception_choice*} => *statement_sequence*

...
[when others =>
statement_sequence]

exception_choice form:
exception_name | others

- Handlers are placed at the end of the block or unit in which they occur

Example:

```
type Age_Type is range 0..125;
type Age_List_Type is array (1..4) of Age_Type;
package Age_IO is new Integer_IO (Age_Type);
use Age_IO;
Age_List : Age_List_Type;
. . .
begin
for Age_Count in 1..4 loop
loop -- loop for repetition when exceptions occur
Except_Blk:
begin -- compound to encapsulate exception handling
Put_Line("Enter an integer in the range 0..125");
Get(Age_List(Age_Count));
exit;
exception
when Data_Error => -- Input string is not a number
Put_Line("Illegal numeric value");
Put_Line("Please try again");
when Constraint_Error => -- Input is < 0 or > 125
Put_Line("Input number is out of range");
Put_Line("Please try again");
end Except_Blk;
end loop; -- end of the infinite loop to repeat input
-- when there is an exception
end loop; -- end of for Age_Count in 1..4 loop
. . .
```

Predefined Exceptions

- CONSTRAINT_ERROR - index constraints, range constraints, etc.
- NUMERIC_ERROR - numeric operation cannot return a correct value (overflow, division by zero, etc.)
- PROGRAM_ERROR - call to a subprogram whose body has not been elaborated
- STORAGE_ERROR - system runs out of heap
- TASKING_ERROR - an error associated with tasks

Evaluation

- Ada was the only widely used language with exception handling until it was added to C++

Exception Handling in C++

- Exceptions are added to C++ in 1990
- Design of exceptions is based on that of Ada, and ML

C++ Exception Handlers

- Exception Handlers Form:

  try {
  -- code that is expected to raise an exception
  }
  catch (*formal parameter*) {
  -- *handler code*
  }
  ...
  catch (*formal parameter*) {
  -- *handler code*
  }

C++ Exception Handling example

```
int main()
{
int a,b;
cout<<“enter a,b values:”;
cin>>a>>b;
    try{
        if(b!=0)
           cout<<“result is:”<<(a/b);
           else
          throw b;
       }
catch(int e)
  {
    cout<<“divide by zero error occurred due
            to b= “ << e;
  }
}
```

The catch Function

- catch is the name of all handlers--it is an overloaded name, so the formal parameter of each must be unique
- The formal parameter need not have a variable
  - It can be simply a type name to distinguish the handler it is in from others
- The formal parameter can be used to transfer information to the handler
- The formal parameter can be an ellipsis, in which case it handles all exceptions not yet handled

Binding Exceptions to Handlers

- Exceptions are all raised explicitly by the statement:
  throw [*expression*];
- The brackets are metasymbols
- A throw without an operand can only appear in a handler; when it appears, it simply re-raises the exception, which is then handled elsewhere
- The type of the expression disambiguates the intended handler

Continuation

- After a handler completes its execution, control flows to the first statement after the last handler in the sequence of handlers of which it is an element

Exception Handling in Java

- Based on that of C++, but more in line with OOP philosophy
- All exceptions are objects of classes that are descendants of the Throwable class

Classes of Exceptions

- The Java library includes two subclasses of Throwable :
  - Error
    - Thrown by the Java interpreter for events such as heap overflow
    - Never handled by user programs
  - Exception
    - User-defined exceptions are usually subclasses of this
    - Has two predefined subclasses, IOException and RuntimeException (e.g., ArrayIndexOutOfBoundsException and NullPointerException

Java Exception Handlers

- Like those of C++, except every catch requires a named parameter and all parameters must be descendants of Throwable
- Syntax of try clause is exactly that of C++, except for the finally clause

Binding Exceptions to Handlers

- Exceptions are thrown with throw, as in C++, but often the throw includes the new operator to create the object as : throw new MyException();

Checked and Unchecked Exceptions

- The Java throws clause is quite different from the throw clause of C++
- Exceptions of class Error and RunTimeException and all of their descendants are called unchecked exceptions; all other exceptions are called checked exceptions
- Checked exceptions that may be thrown by a method must be either:
  - Listed in the throws clause, or
  - Handled in the method

finally clause

- Can appear at the end of a try construct
- Form:

```
finally {
...
}
```

Example:

- Purpose: To specify code that is to be executed, regardless of what happens in the try construct
- A try construct with a finally clause can be used outside exception handling

```
try {
        for (index = 0; index < 100; index++) {
                …
                if (…) {
                        return;
                }  //** end of if
}  //** end of try clause
finally {
        …
}  //** end of try construct
```

Example of Exception Handling in java

```
import java.io.*;
class Test
{
	public static void main(String args[]) throws IOException
	{
		int a[],b,c;
		DataInputStream dis=new DataInputStream(System.in);
		a=new int[5];
		for(int i=0;i<5;i++)
		{
			a[i]=Integer.parseInt(dis.readLine());
		}
		//displaying the values from array
		try{
			for(int i=0;i<7;i++)
			{
				System.out.println(a[i]);
			}
		}
		catch(Exception e)
		{

				System.out.println("The run time error is:"+e);
		}

		finally
	    {

			System.out.print("100% will be executed");
	    }

	}
}
```

O/P: 1 2 3 4 5

1 2 3 4 5
The runtime error is : ArrayIndexOutOfBoundsException: 5
100% wiil be executed

Assertions

- Statements in the program declaring a boolean expression regarding the current state of the computation
- When evaluated to true nothing happens
- When evaluated to false an AssertionError exception is thrown
- Can be disabled during runtime without program modification or recompilation
- Two forms
  - assert *condition*;
  - assert *condition*: *expression*;

Evaluation

- The types of exceptions makes more sense than in the case of C++
- The throws clause is better than that of C++ (The throw clause in C++ says little to the programmer)
- The finally clause is often useful
- The Java interpreter throws a variety of exceptions that can be handled by user program**s**

Introduction to Event Handling

- Event handling is a basic concept of graphical user interfaces.
- An *event* is created by an external action such as a user interaction through a GUI
- The *event handler* is a segment of code that is called in response to an event

Java Swing GUI Components

- Text box is an object of class JTextField
- Radio button is an object of class JRadioButton
- Applet's display is a frame, a multilayered structure
- Content pane is one layer, where applets put output
- GUI components can be placed in a frame
- Layout manager objects are used to control the placement of components

The Java Event Model

- User interactions with GUI components create events that can be caught by event handlers, called *event listeners*
- An event generator tells a listener of an event by sending a message
- An interface is used to make event-handling methods conform to a standard protocol
- A class that implements a listener must implement an interface for the listener
- One class of events is ItemEvent, which is associated with the event of clicking a checkbox, a radio button, or a list item
- The ItemListener interface prescribes a method, itemStateChanged, which is a handler for ItemEvent events
- The listener is created with addItemListener

```
/* RadioB.java
An example to illustrate event handling with interactive
radio buttons that control the font style of a textfield
*/
package radiob;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class RadioB extends JPanel implements
ItemListener {
private JTextField text;
private Font plainFont, boldFont, italicFont,
boldItalicFont;
private JRadioButton plain, bold, italic, boldItalic;
private ButtonGroup radioButtons;
// The constructor method is where the display is initially
// built
public RadioB() {
// Create the test text string and set its font
text = new JTextField(
"In what font style should I appear?", 25);
text.setFont(plainFont);
// Create radio buttons for the fonts and add them to
// a new button group
plain = new JRadioButton("Plain", true);
bold = new JRadioButton("Bold");
italic = new JRadioButton("Italic");
boldItalic = new JRadioButton("Bold Italic");
radioButtons = new ButtonGroup();
radioButtons.add(plain);
radioButtons.add(bold);
radioButtons.add(italic);
radioButtons.add(boldItalic);
// Create a panel and put the text and the radio
// buttons in it; then add the panel to the frame
JPanel radioPanel = new JPanel();
radioPanel.add(text);
radioPanel.add(plain);
radioPanel.add(bold);
radioPanel.add(italic);
radioPanel.add(boldItalic);
add(radioPanel, BorderLayout.LINE_START);
// Register the event handlers
plain.addItemListener(this);
bold.addItemListener(this);
italic.addItemListener(this);
boldItalic.addItemListener(this);
// Create the fonts
plainFont = new Font("Serif", Font.PLAIN, 16);
boldFont = new Font("Serif", Font.BOLD, 16);
italicFont = new Font("Serif", Font.ITALIC, 16);
boldItalicFont = new Font("Serif", Font.BOLD +
Font.ITALIC, 16);
} // End of the constructor for RadioB
// The event handler
public void itemStateChanged (ItemEvent e) {
// Determine which button is on and set the font
```

```
// accordingly
if (plain.isSelected())
text.setFont(plainFont);
else if (bold.isSelected())
text.setFont(boldFont);
else if (italic.isSelected())
text.setFont(italicFont);
else if (boldItalic.isSelected())
text.setFont(boldItalicFont);
} // End of itemStateChanged
// The main method
public static void main(String[] args) {
// Create the window frame
JFrame myFrame = new JFrame(" Radio button
example");
// Create the content pane and set it to the frame
JComponent myContentPane = new RadioB();
myContentPane.setOpaque(true);
myFrame.setContentPane(myContentPane);
// Display the window.
myFrame.pack();
myFrame.setVisible(true);
}
} // End of RadioB
```

Output o`RadioB.java`

POLYMORPHISM AND DYNAMIC BINDING

- A characteristic of object-oriented programming languages is a kind of polymorphism provided by the dynamic binding of messages to method definitions. This is supported by allowing one to define polymorphic variables of the type of the parent class that are also able to reference objects of any of the subclasses of that class.
- Dynamic binding through polymorphic variables is a powerful concept
- The **abstract method** is often called a virtual method
- Any class that includes at least one virtual method is called a **virtual class**.